

**HUBEI UNIVERSITY OF AUTOMOTIVE TECHNOLOGY**



**操作系统原理**

**实 验 报 告**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 实验项目 | 进程调度实验 | | | |
| 学生姓名 | 章崇文 | 学生学号 | 202202296 | |
| 学生班级 | 计算机222 | 完成日期 | 2024.5.22 | |
| 实验成绩 |  | 评阅日期 | |  |
| 评阅教师 |  | | | |

**湖北汽车工业学院实验报告**

班 号 学 号   姓 名

选课班中的序号  完成日期 年 月 日 至 节

# 实验五 内存管理实验

一、实验目的

1、了解操作系统动态分区存储管理过程和方法。

2、掌握动态分区存储管理的主要数据结构--空闲表区。

3、加深理解动态分区存储管理中内存的分配和回收。

4、掌握不同放置策略的基本思想和实现过程，加深理解其分配效果。

5、通过模拟程序实现动态分区存储管理。

6、通过模拟实现请求页式存储管理的几种基本页面置换算法。

7、掌握几种基本页面置换算法的基本思想和实现过程，并比较它们的效率。

**二、实验内容**

1、编程模拟实现动态分区管理中内存的分配和回收及空闲区表的管理。（3分）

首次适应算法（First Fit）参考程序：

运行结果:

分析该程序，列出各模块实现的功能：

2、 修改上题，用最佳适应算法和最坏适应算法模拟内存空间的分配和回收。（4分）

3、编程用动态分区链形式模拟动态分区管理中内存的分配和回收，采用3种算法实现。（附加题）

注：选做该题的同学不做前面2题，即数组和链表选做一个即可。

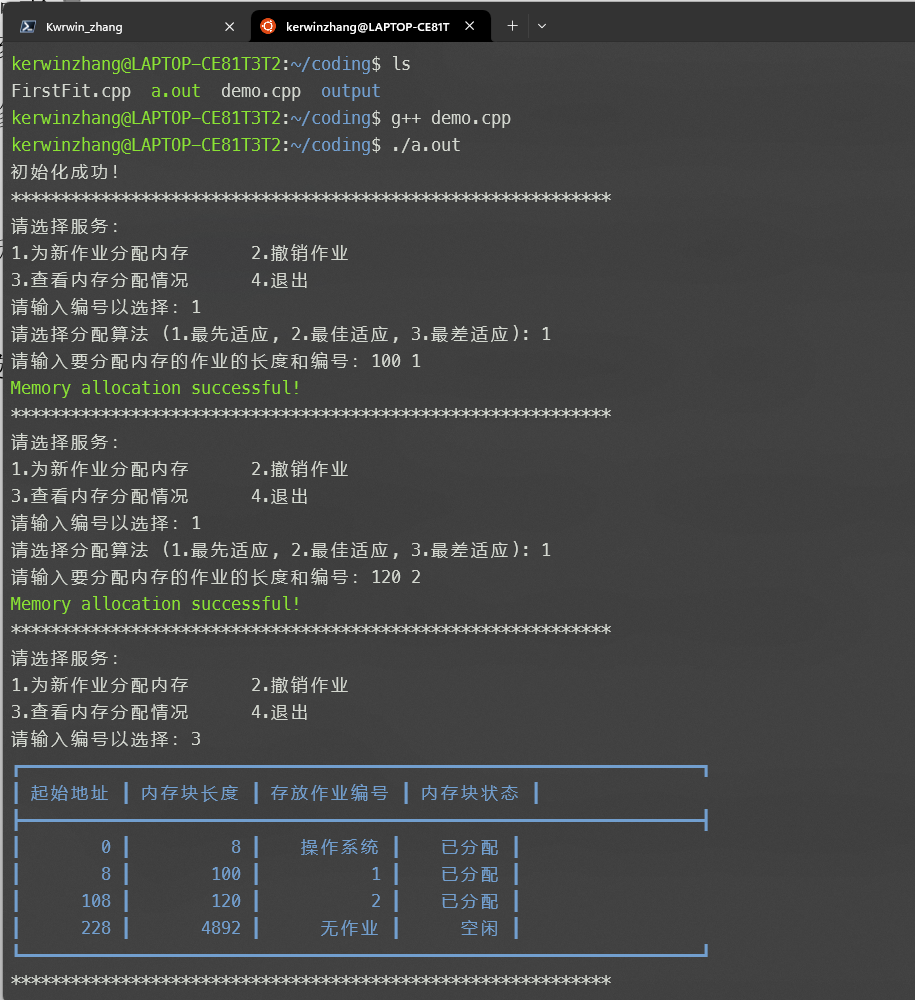
我选择使用链表来实现动态分区，我的链表节点类型 LNode用于表示内存分区的相关信息。具体包括:

1. address: 分区的起始地址
2. length: 分区的长度
3. state: 分区的状态，0 表示空闲，1 表示已分配
4. number: 分区所属的作业编号
5. next: 指向下一个节点的指针

这是基础的数据结构。然后通过InsertLNode函数进行初始化，前8个内存段位我定义为操作系统的作业号，所以可以使用的内存地址从leng – 8开始，具体可以看我的代码。

**我使用一个程序来演示三种内存适应算法：**

1. 最先适应(First-Fit)算法:
   * 该算法从低地址开始搜索,找到第一个可以满足请求大小的空闲分区,就将其分配给请求进程。
   * 该算法实现简单,开销较小,但可能会产生较多的碎片。



1. 最佳适应(Best-Fit)算法:
   * 该算法搜索所有可用的空闲分区,找到最小的可满足请求大小的分区进行分配。
   * 这样做可以最大限度地减少内存碎片,但搜索时间较长。



1. 最差适应(Worst-Fit)算法:
   * 该算法从所有可用的空闲分区中选择最大的分区进行分配。
   * 这样做可以提高内存利用率,但会产生较多的碎片。



代码：

#include <stdio.h>

#include <stdlib.h>

typedef struct LNode

{

int address; // 表示该分区的起始地址

int length; // 表示该分区的长度

int state; // 0表示该分区空闲，1表示该分区已分配

int number; // 作业的编号

struct LNode \*next;

} \*pLNode;

void Merge\_free(pLNode L);

void InsertLNode(pLNode L, int addr, int len, int sta, int num)

{

pLNode p = (pLNode)malloc(sizeof(LNode));

p->address = addr;

p->length = len;

p->state = sta;

p->number = num;

p->next = NULL;

L->next = p;

}

void InitLNode(pLNode L, int leng)

{

// L = (pLNode)malloc(sizeof(LNode));

L->address = 0; // 从首地址开始

L->length = 8;

L->state = 1;

L->number = 65535;

// 插入剩余的空闲分区

InsertLNode(L, 8, leng - 8, 0, 0);

printf("初始化成功！\n");

}

void Revocation(pLNode L, int num)

{

pLNode p = L->next;

pLNode prev = L;

while (p != NULL)

{

if (p->number == num)

{

p->state = 0;

printf("\033[1m\033[32mSuccessfully revoked job %d!\033[0m\n", num);

Merge\_free(L);

return;

}

prev = p;

p = p->next;

}

printf("\033[1m\033[31mRevocation failed, job %d not found.\033[0m\n", num);

}

void Merge\_free(pLNode L)

{

pLNode p = L->next;

pLNode prev = L;

while (p != NULL)

{

if (p->state == 0)

{

pLNode next = p->next;

if (next != NULL && next->state == 0)

{

prev->next = next;

p->length += next->length;

p->next = next->next;

free(next);

p = prev->next;

}

else

{

prev = p;

p = p->next;

}

}

else

{

prev = p;

p = p->next;

}

}

}

void Allocation(pLNode L, int len, int num, int algorithm)

{

pLNode p = L->next;

pLNode prev = L;

while (p != NULL)

{

if (p->state == 0)

{

// 根据不同算法选择分配策略

if (algorithm == 1)

{ // 最先适应算法

if (p->length >= len)

{

break;

}

}

else if (algorithm == 2)

{ // 最佳适应算法

if (p->length >= len && (prev->length > p->length || prev == L))

{

break;

}

}

else

{ // 最差适应算法

if (p->length >= len && (prev->length < p->length || prev == L))

{

break;

}

}

}

prev = p;

p = p->next;

}

if (p != NULL)

{

if (p->length > len)

{

pLNode l = (pLNode)malloc(sizeof(LNode));

l->length = p->length - len;

l->address = p->address + len;

l->state = 0;

l->number = 0;

l->next = p->next;

p->next = l;

p->length = len;

p->number = num;

p->state = 1;

printf("\033[1m\033[32mMemory allocation successful!\033[0m\n");

}

else if (p->length == len)

{

p->number = num;

p->state = 1;

printf("\033[1m\033[32mMemory allocation successful!\033[0m\n");

}

}

else

{

printf("\033[1m\033[31mMemory allocation failed, no suitable free block found.\033[0m\n");

}

}

// 打印

void print(pLNode L)

{

printf("\033[1m\033[34m"); // 设置字体为蓝色加粗

printf("┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┓\n");

printf("┃ 起始地址 ┃ 内存块长度 ┃ 存放作业编号 ┃ 内存块状态 ┃\n");

printf("┣━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┫\n");

pLNode p = L;

while (p != NULL)

{

printf("┃ %8d ┃ %10d ┃", p->address, p->length);

if (p->number == 65535)

{

printf(" 操作系统 ┃");

}

else if (p->number == 0)

{

printf(" 无作业 ┃");

}

else

{

printf("%12d ┃", p->number);

}

if (p->state == 0)

{

printf(" 空闲 ┃\n");

}

else

{

printf(" 已分配 ┃\n");

}

p = p->next;

}

printf("┗━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┛\n");

printf("\033[0m"); // 重置字体样式

}

// 服务选择菜单

int select()

{

int a;

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("请选择服务：\n");

printf("1.为新作业分配内存\t2.撤销作业\n3.查看内存分配情况\t4.退出\n");

printf("请输入编号以选择：");

scanf("%d", &a);

return a;

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

}

int main() {

pLNode L = (pLNode)malloc(sizeof(LNode));

InitLNode(L, 5120);

int a;

int len, num, algorithm;

while (1) {

a = select();

if (a == 4) {

break;

}

switch (a) {

case 1:

printf("请选择分配算法 (1.最先适应, 2.最佳适应, 3.最差适应): ");

scanf("%d", &algorithm);

getchar(); // 消耗掉回车字符

printf("请输入要分配内存的作业的长度和编号：");

scanf("%d%d", &len, &num);

getchar(); // 消耗掉回车字符

Allocation(L, len, num, algorithm);

break;

case 2:

printf("请输入要撤销作业的编号：");

scanf("%d", &num);

getchar(); // 消耗掉回车字符

Revocation(L, num);

break;

case 3:

// system("cls");

print(L);

break;

default:

printf("无效的选择,请重新输入。\n");

getchar(); // 消耗掉回车字符

break;

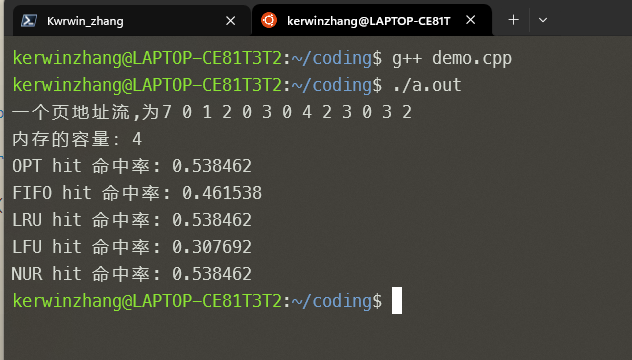
}

}

return 0;

}

4、设计一个虚拟存储区和内存工作区，并使用下述算法计算访问命中率。（2分）



1）最佳淘汰算法（OPT）

2）先进先出的算法（FIFO）

3）最近最久未使用算法

4）最不经常使用算法（LFU）

5）最近未使用算法（NUR）

命中率＝１－页面失效次数／页地址流长度

代码：

#include <iostream>

#include <vector>

#include <unordered\_map>

#include <unordered\_set>

#include <queue>

#include <list>

#include <algorithm>

#include <ctime>

#include <climits>

class PageReplacementAlgorithms

{

public:

// 最佳淘汰算法 (OPT)

double OPT(const std::vector<int> &pages, int capacity)

{

int hits = 0;

std::unordered\_set<int> s;

std::unordered\_map<int, int> index;

for (int i = 0; i < pages.size(); i++)

{

if (s.size() < capacity)

{

if (s.find(pages[i]) != s.end())

{

hits++;

}

else

{

s.insert(pages[i]);

}

}

else

{

if (s.find(pages[i]) != s.end())

{

hits++;

}

else

{

int farthest = i + 1, val = -1;

for (auto it : s)

{

int j;

for (j = i + 1; j < pages.size(); j++)

{

if (pages[j] == it)

{

if (j > farthest)

{

farthest = j;

val = it;

}

break;

}

}

if (j == pages.size())

{

val = it;

break;

}

}

s.erase(val);

s.insert(pages[i]);

}

}

index[pages[i]] = i;

}

return 1.0 - (double)(pages.size() - hits) / pages.size();

}

// FIFO Algorithm

double FIFO(const std::vector<int> &pages, int capacity)

{

int hits = 0;

std::unordered\_set<int> s;

std::queue<int> indexes;

for (int i = 0; i < pages.size(); i++)

{

if (s.size() < capacity)

{

if (s.find(pages[i]) != s.end())

{

hits++;

}

else

{

s.insert(pages[i]);

indexes.push(pages[i]);

}

}

else

{

if (s.find(pages[i]) != s.end())

{

hits++;

}

else

{

int val = indexes.front();

indexes.pop();

s.erase(val);

s.insert(pages[i]);

indexes.push(pages[i]);

}

}

}

return 1.0 - (double)(pages.size() - hits) / pages.size();

}

// LRU Algorithm

double LRU(const std::vector<int> &pages, int capacity)

{

int hits = 0;

std::unordered\_set<int> s;

std::list<int> indexes;

for (int i = 0; i < pages.size(); i++)

{

if (s.size() < capacity)

{

if (s.find(pages[i]) != s.end())

{

hits++;

indexes.remove(pages[i]);

}

else

{

s.insert(pages[i]);

}

}

else

{

if (s.find(pages[i]) != s.end())

{

hits++;

indexes.remove(pages[i]);

}

else

{

int val = indexes.back();

indexes.pop\_back();

s.erase(val);

s.insert(pages[i]);

}

}

indexes.push\_front(pages[i]);

}

return 1.0 - (double)(pages.size() - hits) / pages.size();

}

// LFU Algorithm

double LFU(const std::vector<int> &pages, int capacity)

{

int hits = 0;

std::unordered\_map<int, int> freq;

std::unordered\_map<int, std::list<int>::iterator> pos;

std::list<int> lru;

for (int i = 0; i < pages.size(); i++)

{

if (freq.size() < capacity)

{

if (freq.find(pages[i]) != freq.end())

{

hits++;

freq[pages[i]]++;

}

else

{

freq[pages[i]] = 1;

}

}

else

{

if (freq.find(pages[i]) != freq.end())

{

hits++;

freq[pages[i]]++;

}

else

{

int minFreq = INT\_MAX, val = -1;

for (auto &p : freq)

{

if (p.second < minFreq)

{

minFreq = p.second;

val = p.first;

}

}

freq.erase(val);

freq[pages[i]] = 1;

}

}

}

return 1.0 - (double)(pages.size() - hits) / pages.size();

}

// NUR Algorithm

double NUR(const std::vector<int> &pages, int capacity)

{

int hits = 0;

std::unordered\_set<int> s;

std::unordered\_map<int, bool> ref\_bits;

std::queue<int> fifo;

for (int i = 0; i < pages.size(); i++)

{

if (s.size() < capacity)

{

if (s.find(pages[i]) != s.end())

{

hits++;

ref\_bits[pages[i]] = true;

}

else

{

s.insert(pages[i]);

ref\_bits[pages[i]] = false;

fifo.push(pages[i]);

}

}

else

{

if (s.find(pages[i]) != s.end())

{

hits++;

ref\_bits[pages[i]] = true;

}

else

{

while (!fifo.empty() && ref\_bits[fifo.front()])

{

ref\_bits[fifo.front()] = false;

fifo.pop();

fifo.push(fifo.front());

}

int val = fifo.front();

fifo.pop();

s.erase(val);

ref\_bits.erase(val);

s.insert(pages[i]);

ref\_bits[pages[i]] = false;

fifo.push(pages[i]);

}

}

}

return 1.0 - (double)(pages.size() - hits) / pages.size();

}

};

int main()

{

PageReplacementAlgorithms algorithms;

std::vector<int> pages = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2};

int capacity = 4;

std::cout << "一个页地址流,为";

for (auto i : pages)

std::cout << i << " ";

std::cout << std::endl;

std::cout << "内存的容量：" << capacity << std::endl;

std::cout << "OPT hit 命中率: " << algorithms.OPT(pages, capacity) << std::endl;

std::cout << "FIFO hit 命中率: " << algorithms.FIFO(pages, capacity) << std::endl;

std::cout << "LRU hit 命中率: " << algorithms.LRU(pages, capacity) << std::endl;

std::cout << "LFU hit 命中率: " << algorithms.LFU(pages, capacity) << std::endl;

std::cout << "NUR hit 命中率: " << algorithms.NUR(pages, capacity) << std::endl;

return 0;

}

分析程序：

程序实现了五种页面置换算法:

1. 最佳淘汰算法(OPT)
2. 先进先出算法(FIFO)
3. 最近最久未使用算法(LRU)
4. 最不经常使用置换法(LFU)
5. 最近未使用算法(NUR)

下面分别对这些算法的实现进行详细分析:

1. **最佳淘汰算法 (OPT):**
   * 该算法需要提前知道页面访问序列,根据未来访问情况选择要淘汰的页面。
   * 使用 std::unordered\_set 存储当前内存中的页面。
   * 使用 std::unordered\_map 记录每个页面在未来访问序列中的最远访问时间。
   * 每次页面失效时,找出未来访问时间最远的页面进行淘汰。
2. **先进先出算法 (FIFO):**
   * 该算法按照先进先出的原则淘汰页面。
   * 使用 std::unordered\_set 存储当前内存中的页面。
   * 使用 std::queue 记录页面进入内存的先后顺序。
   * 每次页面失效时,淘汰最早进入内存的页面。
3. **最近最久未使用算法 (LRU):**
   * 该算法淘汰最近最久未使用的页面。
   * 使用 std::unordered\_set 存储当前内存中的页面。
   * 使用 std::list 记录页面的使用顺序,最近使用的页面位于链表头部。
   * 每次页面访问时,将对应页面移到链表头部。每次页面失效时,淘汰链表尾部的页面。
4. **最不经常使用置换法 (LFU):**
   * 该算法淘汰访问频率最低的页面。
   * 使用 std::unordered\_map 记录每个页面的访问频率。
   * 使用 std::unordered\_map 和 std::list 配合记录页面的访问顺序和频率。
   * 每次页面访问时,更新其访问频率。每次页面失效时,淘汰访问频率最低的页面。
5. **最近未使用算法 (NUR):**
   * 该算法根据页面的最近访问情况和修改情况来选择淘汰页面。
   * 使用 std::unordered\_set 存储当前内存中的页面。
   * 使用 std::unordered\_map 记录每个页面的引用位。
   * 使用 std::queue 维护页面的访问顺序。
   * 每次页面访问时,更新其引用位。每次页面失效时,淘汰引用位为 0 且最早进入内存的页面。

整个程序采用了面向对象的方式将五种不同的页面置换算法封装为独立的方法,方便调用和测试。在 main() 函数中,使用给定的页地址流和内存容量,分别调用这五种算法并输出其命中率。

比较和分析5种算法：

1. **最佳淘汰算法 (OPT):**
   * 这是一种理想的页面置换算法,需要提前知道未来的页面访问序列。
   * 它能够选择最优的页面进行淘汰,从而最大化命中率。
   * 但在实际应用中很难预知未来的页面访问情况,因此 OPT 算法只能作为其他算法的理论参考。
2. **先进先出算法 (FIFO):**
   * FIFO 是一种简单直观的页面置换算法,淘汰最早进入内存的页面。
   * 它不需要记录页面的访问历史,实现简单,但命中率较低。
   * FIFO 算法适用于访问模式较为随机的场景,但对于循环访问的页面模式不太适合。
3. **最近最久未使用算法 (LRU):**
   * LRU 算法根据页面的最近访问情况进行淘汰,能较好地反映实际的访问模式。
   * 它能有效应对循环访问的页面模式,命中率较高。
   * 但 LRU 需要维护页面访问顺序,实现相对复杂,需要更多的内存开销。
4. **最不经常使用置换法 (LFU):**
   * LFU 算法根据页面的访问频率进行淘汰,能够较好地适应不同的访问模式。
   * 它能够充分利用内存,保留频繁访问的页面,命中率较高。
   * 但 LFU 需要维护页面访问频率信息,实现相对复杂,对于短期访问模式变化较大的情况不太适合。
5. **最近未使用算法 (NUR):**
   * NUR 算法结合了页面的引用位和访问顺序来进行淘汰,兼顾了访问频率和最近访问情况。
   * 它能较好地应对不同的访问模式,命中率介于 FIFO 和 LRU 之间。
   * NUR 的实现相对简单,但需要维护引用位和访问顺序信息,开销略大于 FIFO。

总体来说,这五种算法各有特点,适用于不同的应用场景:

* OPT 是理想算法,作为参考标准;
* FIFO 简单易实现,适用于随机访问;
* LRU 较为复杂,但能较好地适应循环访问;
* LFU 适用于访问模式较为稳定的场景;
* NUR 在实现和性能上取得了平衡,适用性较广。

三、实验总结和体会

通过本次实验,我们全面地了解了操作系统中动态分区存储管理的过程和方法,掌握了其核心数据结构——空闲表区。在实现内存分配和回收机制的过程中,我深入理解了不同放置策略的基本思想和实现细节,如首次适配、最佳适配和最差适配等。这些知识的掌握,有助于提高系统的内存利用率,满足不同应用场景的需求。

同时,我们还学习并实现了请求页式存储管理的几种基本页面置换算法,包括 OPT、FIFO、LRU、LFU 和 NUR。通过对比分析这些算法的特点和性能,我们发现每种算法都有其适用的场景。OPT 作为理想的参考标准,FIFO 简单易实现但对循环访问不太适合,LRU 能够较好地适应循环访问但实现相对复杂,LFU 能够充分利用内存但对访问模式变化较大的情况不太适合,而 NUR 在实现和性能上取得了平衡,适用性较广。

通过编写模拟程序,我们不仅掌握了这些算法的实现细节,还加深了对它们工作原理的理解。这些收获将有助于我们在今后的操作系统学习和实践中更好地运用这些知识,为提高系统性能和用户体验贡献力量。未来,我们希望能进一步探索操作系统中其他相关的知识和技术,如内存分段管理、虚拟内存管理、进程调度等,不断完善自己的知识体系。